Building Regular Registers with Rational Malicious Servers and Anonymous Clients

Antonella Del Pozzo[1], Silvia Bonomi[1], Riccardo Lazzeretti[1] and Roberto Baldoni[1 2]

[1] Research Center of Cyber Intelligence and Information Security (CIS), IT
Dept. of Computer and System Sciences "Antonio Ruberti", Sapienza Università di Roma, IT
{delpozzo, bonomi, lazzeretti, baldoni}@dis.uniroma1.it
[2] CINI Cybersecurity National Laboratory, Italy

Abstract. The paper addresses the problem of emulating a regular register in a synchronous distributed system where clients invoking read() and write() operations are anonymous while server processes maintaining the state of the register may be compromised by rational adversaries (i.e., a server might behave as *rational malicious Byzantine* process). We first model our problem as a Bayesian game between a client and a rational malicious server where the equilibrium depends on the decisions of the malicious server (behave correctly and not be detected by clients vs returning a wrong register value to clients with the risk of being detected and then excluded by the computation). We prove such equilibrium exists and finally we design a protocol implementing the regular register that forces the rational malicious server to behave correctly.
Keywords: Regular Register, Rational Malicious Processes, Anonymity, Bayesian Game.

1 Introduction

To ensure high service availability, storage services are usually realized by replicating data at multiple locations and maintaining such data consistent. Thus, replicated servers represent today an attractive target for attackers that may try to compromise replicas correctness for different purposes, such as gaining access to protected data, interfering with the service provisioning (e.g. by delaying operations or by compromising the integrity of the service), reducing service availability with the final aim to damage the service provider (reducing its reputation or letting it pay for the violation of service level agreements), etc. A compromised replica is usually modeled trough an arbitrary failure (i.e. a Byzantine failure) that is made transparent to clients by employing Byzantine Fault Tolerance (BFT) techniques. Common approaches to BFT are based on the deployment of a sufficiently large number of replicas to tolerate an estimated number f of compromised servers (i.e. BFT replication). However, this approach has a strong limitation: a smart adversary may be able to compromise more than f replicas in long executions and may get access to the entire system when the attack is sufficiently long. To overcome this issue, Sousa et al. designed the *proactive-reactive recovery* mechanism [22]. The basic idea is to periodically reconfigure the set of replicas to rejuvenate servers that may be under attack (proactive mode) and/or when a failure is detected (reactive mode).

This approach is effective in long executions but requires a fine tuning of the replication parameters (upper bound f on the number of possible compromised replicas in a given period, rejuvenation window, time required by the state transfer, etc...) and the presence of secure components in the system. In addition, it is extremely costly during good periods (i.e. periods of normal execution) as a high number of replicas must be deployed independently from their real need. In other words, the system pays the cost of an attack even if the attack never takes place.

In this paper, *we want to investigate the possibility to implement a distributed shared variable (i.e. a register) without making any assumption on the knowledge of the number of possible compromised replicas*, i.e. without relating the total number of replicas n to the number of possible compromised ones f. To overcome the impossibility result of [5,19], we assume that (i) clients preserve their privacy and do not disclose their identifiers while interacting with server replicas (i.e. anonymous clients) and (ii) at least one server is always alive and never compromised by the attacker. We first model our protocol as a game between two parties, a client and a rational malicious server (i.e. a server controlled by rational adversaries) where each rational malicious server gets benefit by two conflicting goals: (i) it wants to have continuous access to the current value of the register and, (ii) it wants to compromise the validity of the register returning a fake value to a client. However, if the rational malicious server tries to accomplish goal (ii) it could be detected by a client and it could be excluded from the computation, precluding it to achieve its first goal. We prove that, under some constraints, an equilibrium exists for such game. In addition, we design some distributed protocols implementing the register and reaching such equilibrium when rational malicious servers privilege goal (i) with respect to goal (ii). As a consequence, rational malicious servers return correct values to clients to avoid to be detected by clients and excluded by the computation and the register implementation is proved to be correct.

The rest of the paper is organized as follows: Section 2 discusses related works, Section 3 and Section 4 introduce respectively the system model and the problem statement. In Section 5 we model the problem as a Bayesian game and in Section 6 we provide a protocol matching the Bayesian Nash Equilibrium that works under some limited constraints, while in Section 7 we presents two variants of the protocol that relax the constraints , at the expense of some additional communications between the clients or protocol complexity increase. Finally, Section 8 presents a discussion and future work.

2 Related Work

Building a distributed storage able to resist arbitrary failures (i.e. Byzantine) is a widely investigated research topic. The Byzantine failure model captures the most general type of failure as no assumption is made on the behavior of faulty processes. Traditional solutions to build a Byzantine tolerant storage service can be divided into two categories: *replicated state machines* [20] and *Byzantine quorum systems* [5,17,18,19]. Both the approaches are based on the idea that the state of the storage is replicated among processes and the main difference is in the number of replicas involved simultaneously in the state maintenance protocol. Replicated state machines approach requires that every non-faulty replica receives every request and processes requests in the same order be-

fore returning to the client [20] (i.e. it assumes that processes are able to totally order requests and execute them according to such order). Given the upper bound on the number of failures f, the replicated state machine approach requires only $2f + 1$ replicas in order to provide a correct register implementation. Otherwise, Byzantine quorum systems need just a sub-set of the replicas (i.e. *quorum*) to be involved simultaneously. The basic idea is that each operation is executed by a quorum and any two quorums must intersect (i.e. members of the quorum intersection act as witnesses for the correct execution of both the operations). Given the number of failures f, the quorum-based approach requires at least $3f+1$ replicas in order to provide a correct register implementation in a fully asynchronous system [19]. Let us note that, in both the approaches, the knowledge of the upper bound on faulty servers f is required to provide deterministic correctness guarantees. In this paper, we follow an orthogonal approach. We are going to consider a particular case of byzantine failures and we study the cost, in terms of number of honest servers, of building a distributed storage (i.e. a register) when clients are anonymous and have no information about the number of faulty servers (i.e. they do not know the bound f). In particular, the byzantine processes here considered deviate from the protocol by following a strategy that brings them to optimize their own benefits (i.e., they are *rational*) and such strategy has the final aim to compromise the correctness of the storage (i.e., they are *malicious*). In [16], the authors presented Depot, a cloud storage system able to tolerate any number of Byzantine clients or servers, at the cost of a weak consistency semantics called *Fork-Join-Causal consistency* (i.e., a weak form of causal consistency).

Another different solution can rely on Proactive Secret Sharing [26]. Secret Sharing [27] guarantees that a secret shared by a client among n parties (servers) cannot be obtained by an adversary corrupting no more than f servers. Moreover, if no more than f servers are Byzantines, the client can correctly recover the secret from the shares provided by any $f + 1$ servers. Recent Proactive Secret Sharing protocols, e.g. [28], show that Secret Sharing can be applied also to synchronous networks. Even if Proactive Secret Sharing can guarantee the privacy of the data (this is out of the scope of the paper) against up to $f = n - 2$ passive adversaries, the solution has some limitations. First fo all, clients are not able to verify whether the number of Byzantines exceeds f and hence understand if the message obtained is correct. Secondly, Secret Sharing protocols operating in a synchronous distributed system with Byzantines (active adversaries) correctly work with a small number of Byzantines and have high complexity ($f < n/2-1$ and $\mathcal{O}(n^4)$ in [28]).

In [3], the authors introduced the *BAR (Byzantine, Altruistic, Rational) model* to represent distributed systems with heterogeneous entities like peer-to-peer networks. This model allows to distinguish between Byzantine processes (arbitrarily deviating from the protocol, without any known strategy), altruistic processes (honestly following the protocol) and rational processes (may decide to follow or not the protocol, according to their individual utility). Under the BAR model, several problems have been investigated (e.g. reliable broadcast [7], data stream gossip [14], backup service through state machine replication [3]). Let us note that in the BAR model the utility of a process is measured through the cost sustained to run the protocol. In particular, each step of the algorithm (especially sending messages) has a cost and the objective of any rational

process is to minimize its global cost. As a consequence, rational *selfish* processes deviate from the protocol just by skipping to send messages, if not properly encouraged by some reward. In contrast with the BAR model, in this paper we consider malicious rational servers that can deviate from the protocol with different objectives, benefiting from preventing the correct protocol execution rather than from saving messages.

More recently, classical one-shot problems as leader election [1,2], renaming and consensus [2] have been studied under the assumption of rational agents (or rational processes). The authors provide algorithms implementing such basic building blocks, both for synchronous and asynchronous networks, under the so called *solution preference* assumption i.e., agents gain if the algorithm succeeds in its execution while they have zero profit if the algorithm fails. As a consequence, processes will not deviate from the algorithm if such deviation interferes with its correctness. Conversely, the model of rational malicious processes considered in this paper removes implicitly this assumption as they are governed by adversaries that get benefit when the algorithm fails while in [1,2] rational processes get benefit from the correct termination of the protocol (i.e. they are selfish according with the BAR model).

Finally, the model considered here can be seen as a particular case of BAR where rational servers take malicious actions, with the application similar to the one considered in [3]. However, in contrast to [3], we do not assume any trusted third party to identify users, we assume that clients are anonymous (e.g., they are connected through the Tor anonymous network [23]), and we investigate the impact of this assumption together with the rational model. To the best of our knowledge, this is the first paper that analyzes how the anonymity can help in managing rational malicious behaviors.

3 System Model

The distributed system is composed by a set of n servers implementing a distributed shared memory abstraction and by an arbitrary large but finite set of clients $\mathcal{C}$. Servers are fully identified (i.e. they have associated a unique identifier $s_1, s_2 \dots s_n$) while clients are anonymous, i.e. they share the same identifier.

Communication model and timing assumptions. Processes can communicate only by exchanging messages through *reliable* communication primitives, i.e. messages are not created, duplicated or dropped. The system is synchronous in the following sense: all the communication primitives used to exchange messages guarantee a timely delivery property. In particular, we assume that clients communicate with servers trough a *timely* reliable broadcast primitive (i.e., there exists an integer δ, known by clients, such that if a client broadcasts a message m at time t and a server s_i delivers m, then all the servers s_j deliver m by time $t + \delta$). Servers-client and client-client communications are done through "point-to-point" *anonymous timely* channels (a particular case of the communication model presented in [10] for the most general case of homonyms). Considering that clients are identified by the same identifier ℓ, when a process sends a point-to-point message m to an identifier ℓ, all the clients will deliver m. More formally, there exists an integer $\delta' \leq \delta$, known by processes, such that if s_i sends a message m to a client

identified by an identifier ℓ at time t, then all the clients identified by ℓ receive m by time $t + \delta'$ (for simplicity in the paper we assume $\delta = \delta'$).

We assume that channels are *authenticated* ("oral" model), i.e. when a process identified by j receives a message m from a process identified by i, then p_j knows that m has been generated by a process having identifier i.

Failure model. Servers are partitioned into two disjoint sub-sets: *honest* servers and *malicious* servers (*attackers*). Honest servers behave according to the protocol executed in the distributed system (discussed in Section 6) while malicious servers represent entities compromised by an adversary that may deviate from the protocol by dropping messages (omission failures), changing the content of a message, creating spurious messages, exchanging information outside the protocol, etc. Malicious servers are *rational*, i.e. they deviate from the protocol by following a strategy that aims at increasing their own benefit (usually performing actions that may prevent the correct execution of the protocol). We assume that rational malicious servers act independently, i.e. they do not form a coalition and each of them acts for its individual gain.

Servers may also fail by crashing and we identify as *alive* the set of non crashed servers[3]. However, we assume that at least one honest alive server always exists in the distributed system.

4 Regular Registers

A register is a shared variable accessed by a set of processes, i.e. clients, through two operations, namely $\mathsf{read}()$ and $\mathsf{write}()$. Informally, the $\mathsf{write}()$ operation updates the value stored in the shared variable while the $\mathsf{read}()$ obtains the value contained in the variable (i.e. the last written value). Every operation issued on a register is, generally, not instantaneous and it can be characterized by two events occurring at its boundary: an *invocation* event and a *reply* event. These events occur at two time instants (invocation time and reply time) according to the fictional global time.

An operation op is *complete* if both the invocation event and the reply event occur (i.e. the process executing the operation does not crash between the invocation and the reply). Contrary, an operation op is said to be *failed* if it is invoked by a process that crashes before the reply event occurs. According to these time instants, it is possible to state when two operations are concurrent with respect to the real time execution. For ease of presentation we assume the existence of a fictional global clock and the invocation time and response time of operations are defined with respect to this fictional clock. Given two operations op and op', and their invocation event and reply event times ($t_B(op)$ and $t_B(op')$) and return times ($t_E(op)$ and $t_E(op')$), we say that op *precedes* op' ($op \prec op'$) iff $t_E(op) < t_B(op')$. If op does not precede op' and op' does not precede op, then op and op' are *concurrent* ($op||op'$). Given a $\mathsf{write}(v)$ operation, the value v is said to be written when the operation is complete.

In case of concurrency while accessing the shared variable, the meaning of *last written value* becomes ambiguous. Depending on the semantics of the operations, three types of register have been defined by Lamport [15]: *safe*, *regular* and *atomic*. In this paper, we consider a regular register which is specified as follows:

[3] Alive servers may be both honest or malicious.

– Termination: If an alive client invokes an operation, it eventually returns from that operation.
– Validity: A read operation returns the last value written before its invocation, or a value written by a write operation concurrent with it.

Interestingly, safe, regular and atomic registers have the same computational power. This means that it is possible to implement a multi-writer/multi-reader atomic register from single-writer/single-reader safe registers. There are several papers in the literature discussing such transformations (e.g., [6,12,21,24,25] to cite a few). In this paper, we assume that the register is single writer in the sense that no two write() operations may be executed concurrently. However, any client in the system may issue a write() operation. This is not a limiting assumption as clients may use an access token to serialize their writes[4]. We will discuss in Section 8 how this assumption can be relaxed.

5 Modeling the Register protocol as a Game

In a distributed system where clients are completely disjoint from servers, it is possible to abstract any register protocol as a sequence of requests made by clients (e.g. a request to get the value or a request to update the value) and responses (or replies) provided by servers, plus some local computation. If all servers are honest, clients will always receive the expected replies and all replies will always provide the right information needed by the client to correctly terminate the protocol. Otherwise, a compromised server can, according to its strategy, omit to send a reply or can provide bad information to prevent the client from terminating correctly. In this case, in order to guarantee a correct execution, the client tries to detect such misbehavior, react and punish the server. Thus, a distributed protocol implementing a register in presence of rational malicious servers can be modeled as a two-party game between a client and each of the servers maintaining a copy of the register: the client wants to correctly access the register while the server wants to prevent the correct execution of a read() without being punished.

Players. The two players are respectively the client and the server. Each player can play with a different role: servers can be divided into *honest* servers and *malicious* servers while clients can be divided in those asking a *risky request* (i.e., clients able to detect misbehaviors and punish server[5]) and those asking for a *risk-less request* (i.e., clients unable to punish servers).
Strategies. Players' strategies are represented by all the possible actions that a process may take. Clients have just one strategy, identified by $\mathcal{R}$, that is *request information to servers*. Contrarily, servers have different strategies depending on their failure state:

– malicious servers have three possible strategies: (i) $\mathcal{A}$, i.e. *attack the client* by sending back wrong information (it can reply with a wrong value, with a wrong timestamp or both), (ii) $\mathcal{NA}$, i.e. *not attack the client* behaving according to the protocol and (iii) $\mathcal{S}$, i.e. *be silent* omitting the answer to client's requests;

[4] Let us recall that we are in a synchronous system and the mutual exclusion problem can be easily solved also in presence of failures.
[5] Notice that the client ability to detect a server misbehaviors depends on the specific protocol.

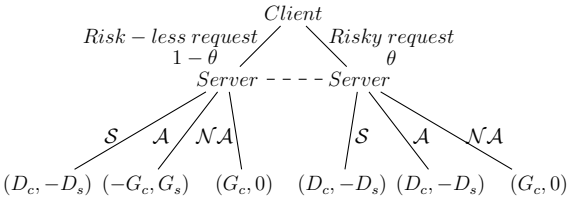

Fig. 1. Extensive form of the game. Dashed line represents the unknown nature of requests from the risk point of view. Outcome pairs refer to client and server gains respectively.

– honest servers have just the $\mathcal{NA}$ strategy.

Let us note that the game between a honest client and a honest server is trivial as they have just one strategy that is to follow the protocol. Thus, in the following we are going to skip this case and we will consider only the game between a client and a rational malicious server.

Utility functions and extensive form of the game. Clients and servers have opposite utility functions. In particular:

– every client increases its utility when it is able to read a correct value from the register and it wants to maximize the number of successful read() operations;
– every server increases its utility when it succeeds to prevent the client from reading a correct value, while it loses when it is detected by the client and it is punished.

In the following, we will denote as G_c the gain obtained by the client when it succeeds in reading, G_s the gain obtained by the server when it succeeds in preventing the client from reading and as D_c the gain of the client when detecting the server and as D_s the loss of the server when it is detected. Such parameters are characteristic of every server and describe its behavior in terms of subjective gains/losses they are able to tolerate. Without loss of generality, we assume that G_c, G_s, D_c and D_s are all greater than 0, that all the servers have the same G_s and D_s[6] and that all the clients have the same G_c and D_c. Fig. 1 shows the extensive form of the game.

The game we are considering is a Bayesian game [11] as servers do not have knowledge about the client role but they can estimate the probability of receiving a risky request or a risk-less request i.e., they have a *belief* about the client role.

We denote as θ (with $\theta \in [0, 1]$) the server belief of receiving a risky request (i.e. the client may detect that the server is misbehaving) and with $1 - \theta$ the server belief of receiving a risk-less request (i.e. the client is not be able to detect that the server is misbehaving).

Analysis of the Bayesian Game. In the following, we are going to analyze the existence (if any) of a *Bayesian Nash Equilibrium* i.e., a Nash Equilibrium[7] computed by considering the players' belief.

Let us note that in our game, clients have just one strategy. Thus, the existence of the

[6] Let us note that if two servers have different values for G_s and D_s, the analysis shown in the following is simply repeated for each server.

[7] Let us recall that a Nash Equilibrium exists when each player selects a strategy and none of the players increases its utility by changing strategy.

equilibrium depends only on the decisions taken by servers according to their utility parameters G_s, D_s and their belief about the nature of a request (i.e., its evaluation of θ).

Let us now compute the expected gain $E()$ of a server s_i while selecting strategies $\mathcal{S}$, $\mathcal{NA}$ and $\mathcal{A}$:

$$E(\mathcal{S}) = (-D_s \times (1-\theta)) + (-D_s \times \theta) = -D_s \quad (1)$$

$$E(\mathcal{NA}) = ((1-\theta) \times 0) + (\theta \times 0) = 0 \quad (2)$$

$$E(\mathcal{A}) = ((1-\theta) \times G_s) - (\theta \times D_s) \quad (3)$$

Lemma 1. *The strategy $\mathcal{S}$ is a dominated strategy.*

It follows that servers have no gain in playing $\mathcal{S}$, whatever the other player does (cf. Lemma 1). In fact, there would be no increment of their utility by playing $\mathcal{S}$ and then we will not consider such strategy anymore.

Let us note that a server s_i would prefer to play $\mathcal{NA}$ (i.e., to behave honestly) with respect to $\mathcal{A}$ (i.e., to deviate from the protocol) when $E(\mathcal{NA}) > E(\mathcal{A})$. Combining equations (3) and (2) we have that a s_i would prefer to play $\mathcal{NA}$ when

$$\frac{G_s}{(G_s + D_s)} > \theta. \quad (4)$$

The parameters G_s and D_s are strictly dependent on the attackers profile (i.e., an attacker for which is more important to stay in the system rather than subvert it or vice versa), thus we can not directly work on them. In the remaining part of the work we propose protocols to tune the θ parameter in such a way that the inequality (4) holds. To this purpose, we derive the following Lemmas:

Lemma 2. *Let s_i be a rational malicious server. If $D_s < G_s$ and $\theta < \frac{1}{2}$ then the best response of s_i is to play strategy $\mathcal{A}$ (i.e. $\mathcal{NA}$ is a dominated strategy).*

Lemma 3. *Let s_i be a rational malicious server. If $D_s > G_s$ and $\theta \geq \frac{1}{2}$ then the best response of s_i is to never play strategy $\mathcal{A}$ (i.e. $\mathcal{NA}$ is a dominant strategy).*

Due to the lack of space, proofs of the previous Lemmas can be found in [9].

6 A Protocol $\mathcal{P}$ for a Regular Register when $D_s \gg G_s$

In this section, we propose a protocol $\mathcal{P}$ implementing a regular register in a synchronous distributed system with anonymous clients and up to $n-1$ malicious rational servers. The protocol works under the assumption that the server loss D_s in case of detection is much higher than its gain G_s obtained when the client fails during a read (i.e. $D_s \gg G_s$[8]). This assumption models a situation where the attacker is much more

[8] More precisely, $\mathcal{P}$ works when $D_s > cG_s$ where c is the estimated number of clients in the system.

interested in having access to data stored in the register and occasionally interfere with the server rather than causing a reduction of the availability (e.g., no termination or validity violation). We will relax this assumption to the simple case $D_s > G_s$ in the next section extending $\mathcal{P}$ in two different ways.

Our protocol $\mathcal{P}$ follows the classical quorum-based approach. When a client wants to write, it sends the new value together with its timestamp to servers and waits for acknowledgments. Similarly, when it wants to read, it asks for values and corresponding timestamps and then it tries to select a value among the received ones. Let us note that, due to the absence of knowledge on the upper bound of malicious processes, it could be impossible for a reader to select a value among those reported by servers and, in addition, the reader may be unable to distinguish well behaving servers from malicious ones. To overcome this issue we leverage on the following observation: the last client c_w writing a value v is able to recognize such value while reading after its write (as long as no other updates have been performed). This makes the writer c_w the only one able to understand which server s_i is reporting a wrong value $v_i \neq v$, detect it as malicious and punish it by excluding s_i from the computation. Thus, the basic idea behind the protocol is to exploit the synchrony of the system and the anonymity of clients to makes the writer indistinguishable from readers and "force" malicious servers to behave correctly.

Let us note that anonymity itself is not enough to make the writer indistinguishable from other clients. In fact, if we consider a naive solution where we add anonymity to a register implementation (e.g., to the one given by Attiya, Bar-Noy and Dolev [4]), we have that servers may exploit the synchrony of the channels to estimate when the end of the write operation occurs and to infer whether a read request may arrive from the writer or from a different client (e.g., when it is received too close to a write request and before the expected end of the write). To this aim, we added in the write() operation implementation some *dummy* read requests. These messages are actually needed to generate message patterns that make impossible to servers to distinguish messages coming from the writer from messages arriving from a different client. As a consequence, received a read request, a server s_i is not able to distinguish if such request is risky (i.e. it comes from the writer) or is risk-less (i.e. it comes from a generic client).

In addition, we added a detection procedure that is executed both during read() and write() operations by any client. In particular, such procedure checks that every server answered to a request and that the reported information are "coherent" with its knowledge (e.g., timestamps are not too old or too new). The detection is done first locally, by exploiting the information that clients collect during the protocol execution, and then, when a client detects a server s_j, it disseminates its detection so that the malicious server is permanently removed from the computation (collaborative detection).

Finally, the timestamp used to label a new written value is updated by leveraging acknowledgments sent by servers at the end of the preceding write() operation. In particular, during each write() operation, servers must acknowledge the write of the value by sending back the corresponding timestamp. This is done on the anonymous channels that deliver such message to all the clients that will update their local timestamp accordingly. As a consequence, any rational server is inhibited from deviating from the protocol, unless it accepts the high risk to be detected as faulty and removed from the system.

```
Init:
(01) replies ← ∅; my_last_val ← ⊥; my_last_ts ← 0; last_ts ← 0;
(02) ack ← ∅; honest ← {s_1, s_2 ... s_n}; writing ← false;
---------------------------------------------------------------
operation read():
(03) if (last_ts = 0)
(04)   then return ⊥;
(05)   else replies ← ∅;
(06)        broadcast READ();
(07)        wait (2δ);
(08)        if (∀ s_i ∈ honest, ∃ < −, ts, val > ∈ replies)
(09)          then broadcast READACK();
(10)               return val;
(11)          else wait (δ);
(12)               if (∀ s_i ∈ honest, ∃ < −, ts, val > ∈ replies)
(13)                 then broadcast READACK();
(14)                      return val;
(15)                 else execute detection(replies_i, R)
(16)                      broadcast READACK();
(17)                      if (∀ s_i ∈ honest, ∃ < −, ts, val > ∈ replies)
(18)                        then return val;
(19)                        else abort ;
(20)                      endif
(21)               endif
(22)        endif
(23) endif
---------------------------------------------------------------
when REPLY(< j, ts, v, ots, ov >) is delivered:
(24) replies ← replies ∪ {< j, ts, v >};
(25) replies ← replies ∪ {< j, ots, ov >};
---------------------------------------------------------------
when DETECTED(s_j) is delivered:
(26) honest ← honest \ {s_j};
```

(a) Client Protocol

```
Init:
(01) val_i ← ∅; ts_i ← 0;
(02) old_val_i ← ⊥; old_ts_i ← 0; reading_i ← 0;
---------------------------------------------------------------
when READ() is delivered:
(03) reading_i ← reading_i + 1;
(04) send REPLY (< i, ts_i, val_i, old_ts_i, old_val_i >);
---------------------------------------------------------------
when READACK() is delivered:
(05) reading_i ← reading_i − 1;
```

(b) Server Protocol

Fig. 2. The read() protocol for a synchronous system.

In the following, we provide a detailed description of the protocol $\mathcal{P}$ shown in Figures 2-4.

The read() **operation (Fig. 2).** When a client wants to read, it first checks if the $last\_ts$ variable is still equal to 0. If so, then there is no write() operation terminated before the invocation of the read() and the client returns the default value $\perp$ (line 04, Fig. 2(a)). Otherwise, c_i queries the servers to get the last value of the register by sending a READ() message (line 06, Fig. 2(a)) and remains waiting for 2δ times, i.e. the maximum round trip message delay (line 07, Fig. 2(a)).
When a server s_i delivers a READ() message, the $reading_i$ counter is increased by one

and then s_i sends a REPLY$(< i, ts_i, val_i, old_ts_i, old_val_i >)$ message containing the current and old values and timestamp stored locally (lines 03 - 04, Fig. 2(b)).
When the reading client delivers a REPLY$(< j, ts, val, ots, ov >)$ message, it stores locally the reply in two tuples containing respectively the current and the old triples with server id, timestamp and corresponding value (lines 24 - 25, Fig. 2(a)). When the reader client is unblocked from the wait statement, it checks if there exists a pair $< ts, val >$ in the $replies$ set that has been reported by all servers it believes honest (line 08, Fig. 2(a)) and, in this case, it sends a READ_ACK() message (line 09, Fig. 2(a)) and it returns the corresponding value (line 10, Fig. 2(a)). Received the READ_ACK() message, a server s_i just decreases by one its $reading_i$ counter (line 05, Fig. 2(b)). Otherwise, a write() operation may be in progress. To check if it is the case, the client keeps waiting for other δ time units and then checks again if a good value exists (lines 11 - 12, Fig. 2(a)). If, after this period, the value is not yet found, it means that some of the servers behaved maliciously. Therefore, the client executes the detection() procedure to understand who is misbehaving (cfr. Fig. 4). Let us note that such procedure cleans up the set of honest servers when they are detected to be malicious. Therefore, after the execution of the procedure, the reader checks for the last time if a good value exists in its $replies$ set and, if so, it returns such value (line 18, Fig. 2(a)); otherwise the special value abort is returned (line 19, Fig. 2(a)). In any case, a READ_ACK() is sent to block the forwarding of new values at the server side (line 16, Fig. 2(a)).

The write() operation (Fig. 3). When a client wants to write, it first sets its $writing$ flag to true, stores locally the value and the corresponding timestamp, obtained incrementing by one the current timestamp stored in $last\_ts$ variable (lines 01 - 02, Fig. 3(a)), sends a WRITE() message to servers, containing the value to be written and the corresponding timestamp (line 03, Fig. 3(a)), and remains waiting for δ time units.
When a server s_i delivers a WRITE(v, ts) message, it checks if the received timestamp is greater than the one stored in the ts_i variable. If so, s_i updates its local variables keeping the current value and timestamp as old and storing the received ones as current (lines 02 - 05, Fig. 3(b)). Contrarily, s_i checks if the timestamp is the same stored locally in ts_i. If this happens, it just adds the new value to the set val_i (line 06, Fig. 3(b)). In any case, s_i sends back an ACK() message with the received timestamp (lines 08, Fig. 3(b)) and forwards the new value if some read() operation is in progress (lines 09, Fig. 3(b)). Delivering an ACK() message, the writer client checks if the timestamp is greater equal than its my_last_ts and, if so, it adds a tuple $< j, ts, - >$ to its ack set (line 16, Fig. 3(a)).
When the writer is unblocked from the wait statement, it sends a READ() message, waits for δ time units and sends another READ() message (lines 06 - 08, Fig. 3(a)). This message has two main objectives: (i) create a message pattern that makes impossible to malicious servers to distinguish a real reader from the writer and (ii) collect values to detect misbehaving servers. In this way, a rational malicious server, that aims at remaining in the system, is inhibited from misbehaving as it could be detected from the writer and removed from the computation. The writer, in fact, executes the detection() procedure both on the ack set and on the $replies$ set collected during the write() (lines 09 - 11, Fig. 3(a)). Finally, the writer sends two READ_ACK() messages to block the forwarding of replies, resets its $writing$ flag to false and returns from the operation

```
operation write(v):
(01) writing ← true; ack ← ∅;
(02) my_last_ts ← last_ts + 1; my_last_val ← v;
(03) broadcast WRITE(< my_last_val, my_last_ts >);
(04) wait(δ);
(05) replies ← ∅;
(06) broadcast READ();
(07) wait(δ);
(08) broadcast READ();
(09) execute detection(ack, A);
(10) wait(δ);
(11) execute detection(replies_i, R);
(12) broadcast READACK();
(13) broadcast READACK();
(14) writing ← false;
(15) return(ok).
```

```
when WRITE_ACK(ts, s_j) is delivered:
(16)  if (ts ≥ my_last_ts) then ack ← ack ∪ {< j, ts, − >} endif
```

```
when ∃ ts such that S = {j|∃ < j, ts', − >∈ ack} ∧ S ⊇ honest:
(17)  if (ts ≥ last_ts) then last_ts ← ts endif
(18)  for each < j, ts', − > ∈ ack such that ts' = ts do ack ← ack \ < j, ts', − > endFor.
```

(a) Client Protocol

```
when WRITE(< val, ts >) is delivered:
(01)  if (ts > ts_i)
(02)    then old_ts_i ← ts_i;
(03)         old_val_i ← val_i;
(04)         ts_i ← ts;
(05)         val_i ← {val};
(06)    else if (ts_i = ts) then val_i ← val_i ∪ {val}; endif
(07)  endIf
(08)  send WRITE_ACK(ts, i);
(09)  if (reading_i > 0) then send REPLY (< i, ts_i, val_i, old_ts_i, old_val_i >) endif.
```

(b) Server Protocol

Fig. 3. write() protocol for a synchronous system.

(lines 12 - 15, Fig. 3(a)).
Let us note that, the execution of a write() operation triggers the update of the $last\_ts$ variable at any client. This happens when in the ack set there exists a timestamp reported by any honest server (lines 17 - 18, Fig. 3(a)).

The detection() **procedure (Fig 4).** This procedure is used by clients to detect servers misbehaviors during the execution of read() and write() operations. It takes as parameter a set (that can be the $replies$ set or the ack set) and a flag that identifies the type of the set (i.e. A for ack, R for replies). In both cases, the client checks if it has received at least one message from any server it saw honest and detects as faulty all the servers omitting a message (lines 01 - 08).
If the set to be checked is a set of ACK() messages, the client (writer) just checks if some server s_j acknowledged a timestamp that is different from the one it is using in the current write() and, if so, s_j is detected as malicious (lines 38 - 42). Otherwise, if the set is the $replies$ set (flagged as R), the client checks if it is running the procedure while it is writing or reading (line 10). If the client is writing, it just updated the state

```
procedure detection(replies_set, set_type):
(01) S = {j|∃ < j, −, − >∈ replies_set};
(02) if (honest ⊈ S)
(03)    then for each s_j ∈ (honest_i \ S) do
(04)                 trigger detect(s_j);
(05)                 honest_i ← honest_i \ {s_j};
(06)                 broadcast DETECTED(s_j);
(07)         endFor
(08) endif
(09) if (set_type = R)
(10)    then if (writing)
(11)           then R = {j|∃ < j, my_last_val, my_last_ts >∈ replies_set};
(12)                if (honest ⊈ R)
(13)                  then for each s_j ∈ (honest_i \ R) do
(14)                           trigger detect(s_j);
(15)                           honest_i ← honest_i \ {s_j};
(16)                           broadcast DETECTED(s_j);
(17)                       endFor
(18)                endIf
(19)           else for each < j, ts, − >∈ replies_set such that ts < last_ts − 1 do
(20)                           trigger detect(s_j);
(21)                           honest ← honest \ {s_j};
(22)                           broadcast DETECTED(s_j);
(23)                endFor
(24)                for each < j, ts, val >∈ replies_set such that ts = my_last_ts do
(25)                           D_i = {v | (∃ < j, ts, val >∈ replies_set) ∧ (ts = my_last_ts)};
(26)                           if ((my_last_val ≠ ⊥) ∧ (my_last_ts = last_ts) ∧ (last_val ∉ D_i))
(27)                             then trigger detect(s_j);
(28)                                  honest ← honest \ {s_j};
(29)                                  broadcast DETECTED(s_j);
(30)                           endif
(31)                endFor
(32)                for each < j, ts, val >∈ replies_set such that ts > last_ts + 1 do
(33)                           trigger detect(s_j);
(34)                           honest_i ← honest_i \ {s_j};
(35)                           broadcast DETECTED(s_j);
(36)                endFor
(37)         endif
(38)    else for each < j, ts, − >∈ replies_set such that ts ≠ my_last_ts do
(39)                  trigger detect(s_j);
(40)                  honest ← honest \ {s_j};
(41)                  broadcast DETECTED(s_j);
(42)         endFor
(43) endif.
```

Fig. 4. detection() function invoked by an anonymous client for a synchronous system.

of the register. Thus, the writer checks that all servers sent back the pair $< v, ts >$ corresponding to the one stored locally in the variables my_last_val and my_last_ts. If someone reported a bad value or timestamp, it is detected as misbehaving (lines 11 - 18). If the client is reading, it is able to detect servers sending back timestamps that are too old (lines 19 - 23) or too new to be correct (lines 32 - 36) or servers sending back the right timestamp but with a wrong value (lines 24 - 31).

Due to the lack of space, the correctness proofs of $\mathcal{P}$ are reported in [9].

7 $\mathcal{P}_{cv}$ and $\mathcal{P}_{hash}$ Protocols for a Regular Register when $D_s \geq G_s$

In the following, we show how to modify the protocol to get $\theta \geq \frac{1}{2}$, when $D_s \geq G_s$. In particular, we propose two possible extensions: the first using a probabilistic collab-

orative detection at the client side (introducing a cost in terms of number of messages needed to run the detection) and the second using a kind of fingerprint to prevent servers misbehavior (introducing a computational cost).

A collaborative detection protocol $\mathcal{P}_{cv}$. The collaborative detection involves all the clients in the detection process and exploits the fact that the last writer remains in the system and it is always able to identify a faulty server. The basic idea is to look for a write witness (i.e., the writer) each time that a reader is not able to decide about the correctness of a value. This solution allows to identify malicious server and to decide and return always a correct value. However, considering that (i) we want to decouple as much as possible servers and client, (ii) this collaborative approach has a cost in terms of messages and (iii) to force rational servers to behave correctly it is sufficient to get $\theta \geq \frac{1}{2}$ (according to Lemma 3), then we use this collaborative approach only with a given probability.

More in details, in $\mathcal{P}_{cv}$ protocol, when a reader does not collect the same value from all servers it flips a coin to decide if running the collaborative detection or not. If the outcome is 1, then it broadcasts to all the other clients the timestamps collected during the read operation and waits that some writer acknowledge them. When a client receives a check timestamp request, it checks if it corresponds to its last written value and if so, it replies with such a value so that the reader can double-check information provided by servers. If there is no match between values and timestamps, then clients are able to detect a faulty server and exclude it from the computation.
The introduction of this probabilistic step in the protocol increases the value of θ to $\frac{1}{2}$. As a consequence, following Lemma 3, any rational server will decide to behave correctly to avoid to be detected.

A fingerprint-based detection protocol $\mathcal{P}_{hash}$. Let us recall that the basic idea behind the detection process is to include inside reply messages (i.e., write acknowledgements or read replies) "enough" information to verify the correctness of the provided information. In particular, in protocol $\mathcal{P}$, servers are required to acknowledge write operations by sending back the corresponding timestamp so that each client is always aware about it and the writer is able to verify that no bad timestamps are sent to clients.

In protocol $\mathcal{P}_{hash}$, the basic idea is to extend $\mathcal{P}$ by including another information i.e., a fingerprint of the value and its timestamp (e.g., its hash), in the write message and in its acknowledgement so that it is always possible for a client to check that servers are replying correctly. More in details, when a client writes, it computes the hash of the value and its corresponding timestamp and attaches such fingerprint to the message. In such way (as for $\mathcal{P}$) when servers acknowledge a write, they send back the correct fingerprint to all clients. Having such information, all clients are potentially able to detect locally if values collected during a read operation are never written values (this can be simply done by computing the hash of the message and compare it with the one received during the last write). However, as in the case of $\mathcal{P}_{cv}$, this detection has a cost and, to get $\theta \geq \frac{1}{2}$ it is sufficient that this is done with a certain probability. Thus, when a reader does not collect the same value from all servers, it flips a coin and if the outcome is 1 then it computes the hash of the messages it delivered and compares them with the hashes it knows to be associated to a specific timestamp. The introduction of this

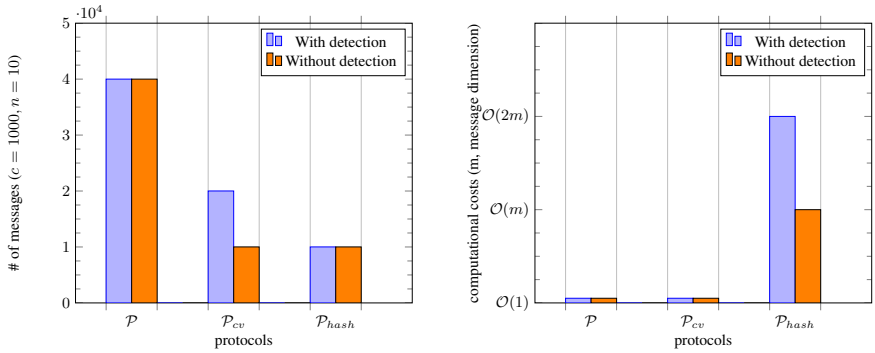

Fig. 5. Qualitative analysis of protocols with respect to their message complexity (left figure) and computational complexity (right figure). For the message complexity we consider a system where the number of servers is $n = 10$ and the number of clients is $c = 1000$. For the computational complexity we consider the cost with respect to the message size m.

step is enough to get $\theta = \frac{1}{2}$ and to prevent rational servers deviating from the protocol. Notice that, as for $\mathcal{P}_{cv}$, the employment of the random coin has a twofold purpose: (i) to provide a solution for $D_s \geq G_s$, for which it is enough to have $\theta \geq \frac{1}{2}$ and (ii) to avoid to always perform the costly detection operation.

Due to the lack of space, proofs for the correctness of $\mathcal{P}_{cv}$ and $\mathcal{P}_{hash}$ protocols are sketched in the [9].

Trade offs. Figure 5 shows a qualitative comparison of the three proposed protocols in terms of message complexity and computational cost. In particular, we compare the cost of the protocols both in presence and absence of a server attack (i.e., when the detection is necessary or not). As we can see, $\mathcal{P}$ requires the highest number of messages and such number does not depend on the real need of doing detection but it is rather required to mask the type of operation that a client is doing and to make indistinguishable real read messages from dummy ones. Concerning its computational cost, it is constant since it does not depend on the message size.
In $\mathcal{P}_{cv}$ it is possible to save the dummy read messages as we do not need anymore to mask the message pattern but we need to pay the cost of the collaborative detection, if it is needed. In fact, if a reader is not able to decide a value, it needs to send messages to contact all the other clients (higher message complexity in case of server misbehaviour). Concerning the computational cost, it is not affected by the detection. Conversely, $\mathcal{P}_{hash}$ exhibits the dual behaviour: message complexity is not affected by server misbehaviour but the computational cost is impacted by the need of detection.

Thus, we can conclude saying that $\mathcal{P}$ is a pessimistic protocol and is the most expensive one but it allows to maintains clients and servers completely decoupled. Contrarily, $\mathcal{P}_{cv}$ and $\mathcal{P}_{hash}$ are optimistic as they perform lightweight operations and, if needed, they perform an heavy detection (with a high message cost in the case of $\mathcal{P}_{cv}$ and a high computational cost in case of $\mathcal{P}_{hash}$).

8 Conclusion

This paper addresses the problem of building a regular register in a distributed system where clients are anonymous and servers maintaining the register state may be rational malicious processes. We have modelled our problem as a two-parties Bayesian game and we designed distributed protocols able to reach the Bayesian Nash Equilibrium and to emulate a regular register when the loss in case of detection is greater than the gain obtained from the deviation (i.e. $D_s > G_s$). To the best of our knowledge, our protocols are the first register protocols working in the absence of knowledge on the number of compromised replicas.

The protocols rely on the following assumptions: (i) rational malicious servers act independently and do not form a coalition, (ii) the system is synchronous, (iii) clients are anonymous and (iv) write operations are serialised.

As future works, we are investigating how to solve the same problem under weaker synchrony assumption or in the case an attacker controls a coalition of processes. Addressing these points is actually far from be trivial. Considering a fully asynchronous system, in fact, makes impossible to use our punishment mechanism as clients are not able to distinguish alive but silent servers from those crashed. Additionally, when the attacker is able to compromise and control a coalition of processes, the model provided in this paper is no more adeguate and we are studying if and how it is possible to define a *Bayesian Coalitional Game* [13] for our problem and if an equilibrium can be reached in this case.

Acknowledgments

This present work has been partially supported by the EURASIA project, and CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks (www.filierasicura.it) funded by CISCO Systems Inc. and Leonardo SpA.

References

1. Abraham, I., Dolev, D., Halpern, J. Y. *Distributed Protocols for Leader Election: A Game-Theoretic Perspective.* DISC 2013: 61-75
2. Afek, Y., Ginzberg, Y., Landau Feibish, S. and Sulamy, M. *Distributed computing building blocks for rational agents.* PODC 2014: 406-415.
3. Aiyer, A. S., Alvisi, L., Clement, A., Dahlin, M., Martin, J. P., and Porth, C. *BAR fault tolerance for cooperative services*. ACM SIGOPS Operating Systems Review. ACM, 2005. p. 45-58.
4. Attiya, H., Bar-Noy, A., and Dolev, D. *Sharing memory robustly in message-passing systems*. Journal of the ACM 42, 1, 1995, 124-142.
5. Bazzi R. A., *Synchronous Byzantine Quorum Systems*, Distributed Computing 13(1), 45-52, 2000.
6. Chaudhuri S., Kosa M.J. and Welch J., *One-write Algorithms for Multivalued Regular and Atomic Registers.* Acta Informatica, 37:161-192, 2000.

7. Clement, A., Li, H. C., Napper, J., Martin, J., Alvisi, L., Dahlin, M. *BAR primer*, DSN 2008: 287-296
8. Clement, A., Napper, J., Li, H., Martin, J. P., Alvisi, L., and Dahlin, M. *Theory of BAR games*, PODC 2007: 358-359
9. Del Pozzo, A., Bonomi S., Lazzeretti R., and Baldoni R. *Building Regular Registers with Rational Malicious Servers and Anonymous Clients – Extended Version* (Available on line on arXiv), 2017.
10. Delporte-Gallet, C., Fauconnier, H., Tran-The, H. *Uniform Consensus with Homonyms and Omission Failures* ICDCN 2013: 161-175
11. Fudenberg, D., Tirole, J. *Game theory*, 1991. Cambridge, Massachusetts.
12. Haldar S. and Vidyasankar K., *Constructing 1-writer Multireader Multivalued Atomic Variables from Regular Variables.* JACM, 42(1):186-203, 1995.
13. Ieong, S., Shoham, Y. *Bayesian Coalitional Games.* AAAI. 2008: 95-100
14. Li, H. C., Clement, A., Wong, E. L., Napper, J., Roy, I., Alvisi, L., Dahlin, M. *BAR Gossip* OSDI 2006: 191-204
15. Lamport. L., On Interprocess Communication, Part 1: Models, Part 2: Algorirhms, *Distributed Computing*, 1(2):77-101, 1986.
16. Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. *Depot: Cloud storage with minimal trust*, ACM TOCS 29(4), 2011
17. Malkhi D., Reiter M. K. *Byzantine Quorum Systems*, Distributed Computing 11(4), 203-213, 1998.
18. Martin J., Alvisi L., Dahlin M., *Small Byzantine Quorum Systems*, DSN 2002: 374-388.
19. Martin J., Alvisi L., Dahlin M.. *Minimal Byzantine Storage*, DISC 2002.
20. Schneider, F. B. *Implementing fault-tolerant services using the state machine approach: A tutorial*, ACM Computing Surveys 22(4): 299-319, 1990.
21. Singh A.K., Anderson J.H. and Gouda M., *The Elusive Atomic Register.* JACM, 41(2):331-334, 1994.
22. Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., Verissimo, P. *Highly available intrusion-tolerant services with proactive-reactive recovery*, IEEE TPDS 21(4): 452-465, 2010 .
23. The Tor Project `https://www.torproject.org`.
24. Vidyasankar K., *Converting Lamport's Regular Register to Atomic Register.* IPL, 28(6):287-290, 1988
25. Vityani P. and Awerbuch B., *Atomic Shared Register Access by Asynchronous Hardware.* FOCS 1987, 223-243.
26. Ostrovsky, R. and Yung, M., *How to withstand mobile virus attacks.* PODC 1991, 51-59.
27. Shamir, A. *How to share a secret* Comm. of ACM 1979, 612-613.
28. Dolev, S., ElDefrawy, K., Lampkins, J., Ostrovsky, R., and Yung, M. *Proactive secret sharing with a dishonest majority.* SCN 2016, 529-548.
29. Cramer, R. and Damg ard, I. B. *Secure Multiparty Computation* Cambridge University Press 2015.